Patterns of Data Distribution

Workshop on Real-Time, Embedded and Enterprise-Scale Time-Critical Systems

April 2012; Paris

Rick Warren
Dir. Tech. Solutions, Principal Engineer
rick.warren@rti.com
System Lifecycle

1. Build cool **small app**.
2. Needs more capability! **Scale out**.
3. Interop., maintenance, and reuse are hard! **Refactor out platform**.

Repeat until you’re ready to...

4. Build **new capabilities** atop unmodified platform.

© 2012 RTI
Think Different.

Building Applications
• Designed as a unit
• Managed as a unit
• ...To serve a single business function

Building Platforms
• Connect independently developed applications
• Focus on interoperability and heterogeneous versioning
• ...To solve business function(s) different than its constituent parts

We will discuss just one aspect here:
Patterns of data distribution

Mostly from this perspective
Distribution Pattern Overview

 Publish-Subscribe  Point to Point  Request-Reply

* See Hohpe and Woolf, “Enterprise Integration Patterns”, www.eaipatterns.com
Pattern: Publish-Subscribe

Scenario for App Use
• Distributing data

Variation Points
• Topic structure: flat vs. hierarchical
• Subscriber quorum
**Pattern: Point to Point**

**Scenario for App Use**
- Distributing computation

**Variation Points**
- Allowed current consumers
- Message visibility to consumers of other roles
Pattern: Request-Reply

Scenarios for App Use

• Sending / acknowledging commands
  – Called “Command Pattern”
• Pulling data (often large)

Variation Points

• # Repliers
• # Replies per Replier
• Synch. vs. asych.
• RMI vs. messaging API
Middleware Comparison

Middleware standards hard code specific patterns.

- **JMS**
  - Publish-Subscribe
    - TopicPublisher
    - TopicSubscriber
  - Point to Point
    - QueueProducer
    - QueueConsumer/QueueBrowser
  - Request-Reply (idiomatic)
    - Based on JMSReplyTo header

- **DDS**
  - Publish-Subscribe
    - DataWriter
    - DataReader

- **HTTP**
  - Request-Reply (data-centric)
    - Messaging: GET, PUT, POST, DELETE, etc.

- **CORBA**
  - Request-Reply (imperative)
  - Point to Point (partial)
    - oneway call to predefined recipient
Combining Patterns

**Insight:**

- Data is global
- Patterns are local

**Platforms are concerned with this**

**Applications are concerned with this**
Identifying Super-Patterns

• **Build platform with super-patterns**
  
  *Super-pattern*. [noun] Pattern from which all others in same category can be derived, by:
  
  1. Adding constraints *and/or*
  2. Composing it with itself

• **Build applications (1) based on derived patterns or (2) isolate logic from pattern altogether**
  
  – For programming ease of use
  – For correctness
  – For manageability
This is the Super-Pattern

- Someone produces some data
- Some other(s) observe(s) that data

- Apply platform concerns here
  - Interoperability
  - Connectivity
  - Governance
We Can Constrain It

- Apply delivery policy to limit observation based on Receiver role(s)
- *Encompasses concurrent consumers and multiple roles*

• Apply application concerns here
  - Portability
  - Ease of (business logic) development
We Can Compose It

• Request, Reply Channels each instance of super-pattern
• *Encompasses multiple Repliers, multiple Replies, additional producers and consumers*

• Apply application concerns here
  – Portability
  – Ease of (business logic) development
Challenges for Platform Builders

Challenge: System Architecture

Architects and developers need training and support

• Many platform developers were trained as application developers, not systems engineers

• Many people think imperatively: “Do this, then do this, ...”
Challenges for Platform Builders

Challenge: Performance

• General, reconfigurable implementation typically has overhead vs. highly specialized one

• ...But beware premature optimization
Challenges for Platform Builders

Challenge: Application Interface

Option #1: Provide Derived Patterns?
- Application developers will demand many of them!
  - Pub-Sub, Point to Point, Request-Reply
  - Others?
- Pro: Perceived as familiar, easy to use
- Con: Many ways to do same thing; less reconfigurable; heavy work for vendors

Option #2: Separate Apps from Patterns?
- Example (JMS MDB):
  ```java
  void onMessage(
    Message event) {
    // biz logic...
  }
  ```
- Pro: Elegant; one thing to learn; easy to reconfigure
- Con: Requires inversion of control => requires container, retraining
Summary

• From app programming POV, certain problems feel natural to solve with certain patterns

• From platform or systems POV, handling each pattern in independent *ad hoc* ways results in brittle, poorly performing systems

• Solution: Build platform in terms of fundamental *super-patterns*
  1. *Option:* Derive high-level patterns from super-patterns; expose those to apps
  2. *Option:* Invert control; enforce patterns within platform, not apps; make app developers cope
P.S. Exercise for the Reader

Find the Super-Patterns:

• **Topology**
  - Peer-to-Peer
  - Hub and Spoke
  - *etc.*

• **Invocation**
  - Event-Driven
  - Imperative
  - *etc.*